17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 16-05-2020
Evaluator(s): Nishitha R and Vinay Banakar
Submission ID: 206
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Kirthika Gurumurthy	PES1201700230	H
2	Richa	PES1201700688	H
3	Mainaki Saraf	PES1201701002	D
4	Akanksha	PES1201701799	H

Introduction

We have built a fault-tolerant, highly available database as a service for a RideShare application. The users and rides microservices created in the previous assignments use the DBaaS service as opposed to using their own databases. A custom orchestrator serves a flask application which listens to incoming HTTP requests from the users and rides and performs the DB reads and DB writes according to the architecture specified below. We've used AMQP using RabbitMQ as a message broker and have implemented scalability (scaling in and scaling out depending on the number of requests received in a duration of time) and high availability using Zookeeper.

Related work

1. https://www.rabbitmq.com/tutorials/tutorial-two-javascript.html
2. https://kazoo.readthedocs.io/en/latest/basic_usage.html
3. https://kazoo.readthedocs.io/en/latest/api/client.html
4. https://docker-py.readthedocs.io/en/stable/

Algorithm/Design

1. **Users and Rides APIs :**

   The APIs from the assignments were all made to go via the db/read and db/write APIs in the orchestrator. Some additional APIs were written for the rides microservice to get the required information for the existing APIs eg: auto increment of ride number, the last number of the last ride was needed. The users and rides containers were present in different instances. A load balancer distributes the incoming HTTP requests to one of the rides or users instances based on the URL route of the request - /api/v1/users route requests are forwarded to the users instance and requests with all other routes are forwarded to the rides instance.

2. **Orchestrator:**

   The orchestrator has the db/read and db/write APIs along with the worker-related APIs. The db/write API publishes each write request to the write queue and db/read request publishes the read requests to

the read queue. The db/read api also consumes messages from the response queue to return read responses. The orchestrator was present in a separate DBaaS instance.

3. **Master:**

We followed the new specifications, so for the master worker, it was assumed that it would never fail. Thus, the master's container only had the code used to write to the master's db. This worker listens only to the write queue and consumes the messages to make the respected db writes. It also publishes the write messages to sync queue 1.

4. **Slave:**

The slave workers only listens to the read queue and publishes the responses for the consumed read requests to the response queue. It has the code only for the reading. Each slave has a mongo-db associated with it. There is one slave that is started initially.

5. **Syncing:**

The syncing was dealt with as follows:

5.1. Every write request consumed by the master was published to sync queue 1.
5.2. There is a designated worker container called sync-worker that is started during program startup.
5.3. This listens and consumes messages published to the sync queue 1 and writes these messages to a collection called "logs" in its db. Each record has a unique id.
5.4. On the first message consumed, the sync worker starts a background scheduler program that publishes all the records in the "logs" collection as a message to a sync queue 2 which has a fanout type exchange. This happens every second.
5.5. Each slave, when started, has a child process running with the main process of consuming read requests, that listens to this sync queue 2 and writes the messages consumed to its own db.
5.6. The child process is started so that no read request is blocked when the syncing takes place and the syncing continues in the background.

5.7. Each slave has a file logs.txt that stores the id of each mongo-db element received from the sync worker it has added to its db. This file is checked before adding records to the db to avoid data redundancy.

6. Scale Up and Scale Down:

6.1. Each read request to the db/read in the orchestrator is tracked using a counter variable.

6.2. When the first read request is received, a background scheduler program is started that checks this read count every 120 seconds.

6.3. If the number of reads is more such that more slave workers are needed, N slave worker containers along with their respective mongo containers are started where N = required workers-existing workers.

6.4. If the number of reads is less and the required workers are more, the extra slave workers and their respective mongo containers are stopped.

7. Zookeeper:

7.1. For each slave started, an ephemeral Znode is started with the number corresponding to the slave number.

7.2. In the orchestrator, a watch is registered to keep a track of all the running Znodes.

7.3. Everytime the state of a node changes such that the Znode loses the connection, the killed slave number is retrieved and the slave is restarted.

7.4. In order to differentiate between a slave crash and a scale-down process, a signal variable is set on each crash which is checked by the zookeeper before starting a new slave and is reset again to detect the next crash if a slave is restarted.

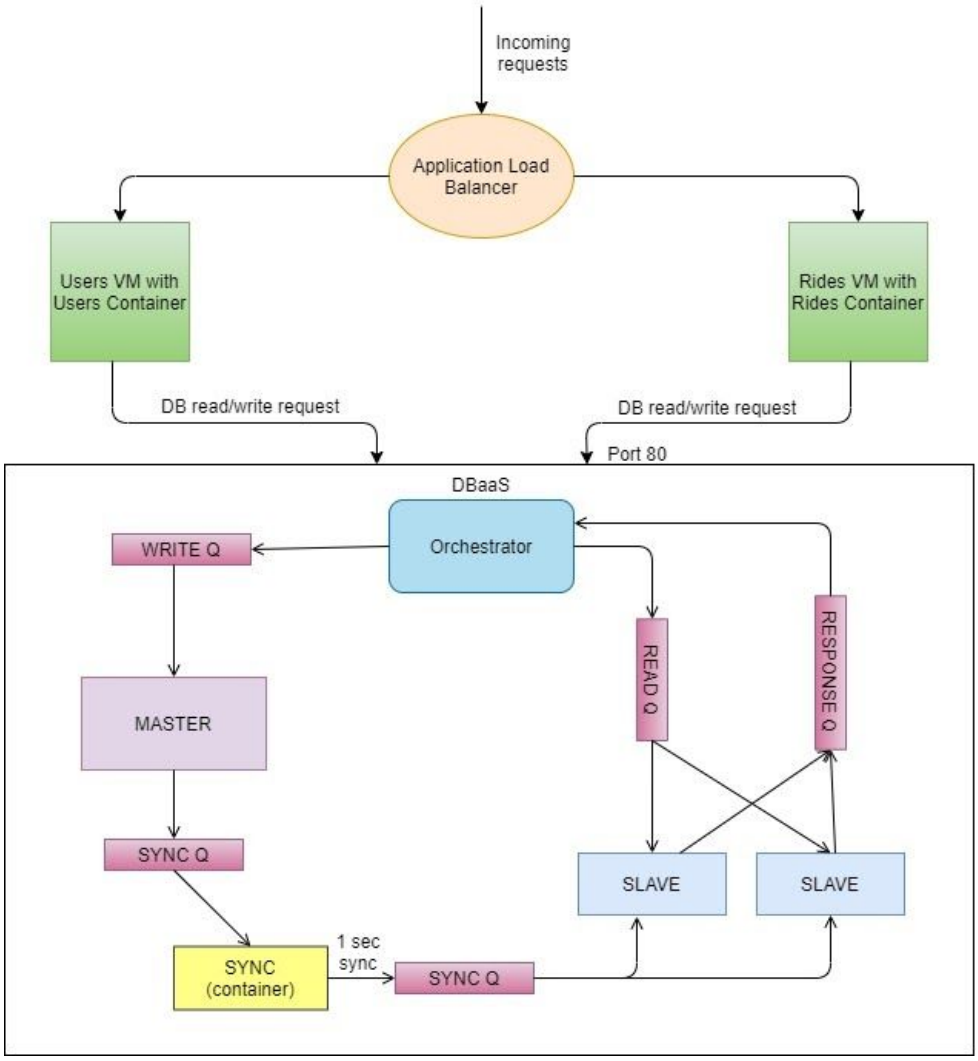

Fig. Architecture

Testing

1. **Problem:**

   The sync worker was syncing every 1 sec, this posed as a problem on the portal as the 1st slave on startup was getting synced after 1 second of the 1st write, but the ride creation api was being hit before 1 sec was completed, thus the user read in the create ride api was not returning a valid response, thus making the api fail.

Solution:

On the first write itself, the message was published directly to sync queue 2 to be written to the slave db instead of going via the background scheduler.

2. **Problem:**

   We had used a SIGKILL signal to kill the slave container in the kill api which worked locally, however, this threw an error while submitting in the portal.

   Solution:

   We changed the docker SDK command to kill( ).

Challenges

1. Syncing was a challenge as we had to figure out a way to sync the workers without blocking read calls and not using only rabbitMQ as it's a messaging channel system, not a database to store all the messages, thus we came up with the above described db based architecture, instead of making the messages persist in the syncQ which would've made the queue run out of memory if too many writes were sent.
2. The syncing was initially made eventually consistent which had to be changed to strictly consistent.
3. Mapping of the slave containers to their respective mongo containers was an issue for which we came up with a nomenclature based on numbering.
4. Scaling to slaves more than 5 consumed all the memory of the AWS instance, for which we had to shift to an instance with more volumes to allow scaling up to a large number of slaves.
5. The flask code reloaded with any change by default so while dynamically creating containers, the names associated with those containers were already present leading to an error, so we had to set the use_reloader parameter in app.run to False.

Contributions

Mainaki:

1. Rides and users apis
2. Orchestrator db/read and db/write apis
3. Read, write and response queues
4. Syncing architecture

Richa:

1. Load Balancer - docker setup for rides and users on two separate instances
2. Crash and worker list APIs
3. Scalability - Scale Down
4. High availability with zookeeper
5. Integration

Kirthika:

1. Docker setup - setting up and building containers, docker-compose, mapping ports
2. Docker SDK - creating containers dynamically
3. Scalability - Scale Up
4. High availability with zookeeper
5. Integration

Akanksha:

1. Rides and users APIs
2. Integration
3. Docker setup
4. Report

Checklist

SNo	Item	Status
1.	Source code documented	
2.	Source code uploaded to private github repository	
3.	Instructions for building and running the code. Your code must be usable out of the box.	